E1(a) \( k = 2; \ x = 2; \) // (because of the conjunct \( 2 \leq k \) in the invariant, we can’t set \( k \) to 1)
// invariant: P1: \( 2 \leq k \leq 10 \) and \( x \) is the product of \( 2..k \)
while (k != 10) {
    x= x * (k+1);
    k= k+1;
}
// postcondition R is: \( x \) is the product of \( 2..10 \)

E1(b) \( k = 2; \ x = 1; \) // (the product of no values is 1; the sum of no values is 0)
// invariant P2: \( 2 \leq k \leq 11 \) and \( x \) is the product of \( 2..(k - 1) \)
while (k != 11) {
    x= x * k;
    k= k + 1;
}
// postcondition R is: \( x \) is the product of \( 2..10 \)

E1(c) \( k = 10; \ x = 10; \)
// invariant: P3: \( 2 \leq k \leq 10 \) and \( x \) is the product of \( k..10 \)
while (k != 2) {
    x= x * (k-1);
    k= k-1;
}
// postcondition R is: \( x \) is the product of \( 2..10 \)

E1(d) \( k = 10; \ x = 1; \)
// invariant: P4: \( 1 \leq k \leq 10 \) and \( x \) is the product of \( (k + 1)..10 \)
while (k != 1) {
    x= x * k;
    k= k-1;
}
// postcondition R is: \( x \) is the product of \( 2..10 \)

E2(a) Note: the conjunct !b in the loop condition is not necessary. It was added later, as an afterthought,
for the following reason. The repetend never falsifies b, so once b becomes true, meaning that some integer
in the range divides n, the loop can terminate. This holds for all four subexercises.
\( k = \) first-1; \( b = \) false;
// P1: first – 1 \( \leq k \leq \) last and \( b = \) "n is divisible by an integer in first..k"
while (!b && k != last) {
    if (n % k == 0)
        b= true;
    k= k+1;
}
// postcondition R: \( b = \) "n is divisible by an integer in first..last"

E2(b) \( k = \) first; \( b = \) false;
// invariant: P2: \( b = \) "n is divisible by an integer in first..(k – 1)"
while (!b && k-1 != last) {
    if (n % k == 0)
        b= true;
    k= k+1;
}
// postcondition R: \( b = \) "n is divisible by an integer in first..last"

E2(c) \( k = \) last + 1;
\( b = \) false;
// invariant: P3: \( b = \) "n is divisible by an integer in k..last"
while (!b && k != first ) {
    if (n%(k-1) == 0)
        b= true;
    k= k – 1;
}
// postcondition R: b = "n is divisible by an integer in first..last"

E2(d)

k= last;
b= false;
// invariant: P4: b = "n is divisible by an integer in k+1..last"
while (!b && k+1 != first ) {
    if (n % k == 0)
        b= true;
    k= k – 1;
}
// postcondition R: b = "n is divisible by an integer in first..last"

E3.
// Precondition: n > 0
int k = 0; int b= 1;
// invariant: 1 ≤ 2**k ≤ n and b = 2**k
while (2*b <= n){
    b= 2*b;
    k= k + 1;
}
// postcondition: 1 ≤ 2**k ≤ n < 2**(k+1)

E4.  
// precondition: x >= 0 and y > 0
int q= 0;
int r= x;
// invariant: x = y * q + r
while (r > y){
    q= q + 1;
    r= r – y;
}

E5.  
// precondition: x > 0 and y > 0 are integers
int b= x;
int c= y;
// invariant: b gcd c = x gcd y
while (b != c){
    /* use whichever property of gcd (given in the exercise description
        that makes progress (decreases b or c) while keeping b and c positive
        and maintaining the invariant. */
    if (b > c)b= b – c;
    else c= c – b;
}

E6.  
// precondition: t is a String and not null.
StringBuffer s= new StringBuffer(t);
int k = 0;
// invariant: s[0..(k-1)] contains no vowels and k != s.length() + 1
while (k != s.length()){ 
    // c= the character at index k, converted to lower case;
    char c= s.charAt(k);
    c= Character.toLowerCase(c);

    // Make progress here by either decreasing s.length()
    // (by removing a character) or increasing k.
    if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
        s.deleteCharAt(k);
    } else {
        k = k + 1;
    }
}
}
    t = s.toString();

E7.  // precondition:  s1.length() = s2.length()
    int k = 0;
    boolean areComplements = true;
    // invariant:  areComplements = “s1[0..(k–1)] is the DNA complement of s2[0..(k–1)]”
    while (areComplements && k != s1.length()){
        char c1 = s1.charAt(k);
        char c2 = s2.charAt(k);
        // code below results in areComplements = 'c1 and c2 are DNA complements'
        if (c1 == 'A' && (c2 != 'T')) { areComplements = false; }
        if (c1 == 'C' && (c2 != 'G')) { areComplements = false; }
        if (c1 == 'G' && (c2 != 'C')) { areComplements = false; }
        if (c1 == 'T' && (c2 != 'A')) { areComplements = false; }
        k = k + 1;
    }
}
    String s_comp = "";
    // invariant: s_comp is the DNA complement of s[0..k-1]
    for (int k = 0; k < s.length(); k = k + 1) {
        if (s.charAt(k) == 'A') { s_comp = s_comp + 'T'; }
        else if (s.charAt(k) == 'T') { s_comp = s_comp + 'A'; }
        else if (s.charAt(k) == 'G') { s_comp = s_comp + 'C'; }
        else { s_comp = s_comp + 'G'; }
    }
    // s_comp is the DNA complement of a s

E9.  // precondition:  n > 0
    int b = 0;
    int a = 1;
    int i = 1;
    // invariant:  a = f[i] and b = f[i1]
    while (i < n) {
        temp = a + b;
        b = a;
        a = temp;
        i = i + 1;
    }
    // postcondition:  i = n (and, therefore, a = fn)

E10.  sum = 0;
    // invariant:  sum = sum of integers that are already read from file
    while (in.available) {
        i = in.readInt();
        sum = sum + i;
    }
    // postcondition:  sum = sum of the integers in the file

E11.  int countOdd = 0;
    int countEven = 0;
// in: countOdd and countEven contain the number of odd and even integers already read from file
while (in.available) {
    int i = in.readInt();
    if (i % 2 == 0)
        countEven = countEven + 1;
    else
        countOdd = countOdd + 1;
}

// postcondition: countOdd and countEven contain the number of odd and even integers in the file

E12. // precondition: n >= 0;
int i = 0;
// invariant: balance contains the balance in the account after i years
while (i < n) {
    balance = balance + balance * rate;
    i = i + 1;
}

// postcondition: balance contains the balance in the account after n years

E13. double e = 1;
int k = 0;
double tk = 1;
// invariant: e = 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + ... + 1/k! and tk = 1/k!
while (tk >= 1E-14) {
    k = k + 1;
    tk = tk/k;
    e = e + tk;
}

E14. The loop below required 20001 iterations to find the approximation 3.141597653564762 to pi = 3.141592653589793. That's far too long, and this is not a good way to calculate pi.

    int k = 0;
    double t = 4;
    double pi = t;
    int sgn = 1;
    // invariant: pi = 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - ... + (-1)**k*4/(2k+1) and
    // t = 4/(2*k+1) and
    // sgn = (-1)**k
    while (t >= .00001) {
        k = k + 1;
        t = 4.0/(2*k+1);
        sgn = -sgn;
        pi = pi + sgn*t;
    }

E15. This loop is preferable to that of E14 because it took only 10 iterations to stop with the same stopping conditions

    double c = 2.0*Math.sqrt(3);
    int k = 0;
    double term = c;
    int t = 1;
    double pi = c;
    // inv: pi = c*(1*3**0) - c/(3*3**1) + c/(5*3**2) - c/(7*3**3) + ...
    // + (-1)**k * c * / ((2*k+1) * 3**k) and
    // t = 3**k and
    // term = c * / ((2*k+1) * 3**k)
while (term >= .00001) {
    k= k+1;
    t= t*3;
    term= c/((2*k+1)*t);
    if (k%2 == 1) pi= pi - term;
    else pi= pi + term;
}

E16. int ndarts= 10000; // number of darts to throw
int k= 0;
int nhits= 0;
java.util.Random rand = new java.util.Random(System.currentTimeMillis());
/* invariant: nhits = number of hits after k darts thrown */
while (k < ndarts) {
    double x = 2 * rand.nextDouble() - 1;
    double y = 2 * rand.nextDouble() - 1;
    if (x*x + y*y <= 1)
        nhits= nhits + 1;
    k= k + 1;
}
/* postcondition: nhits = no. of hits after darts thrown and k = ndarts */
double pi = 4.0 * nhits / ndarts;

E17. int n= 0;
// inv: n is the number of times 'a' occurs in s[0..k-1]
for (int k= 0; k != s.length(); k= k+1) {
    if (s.charAt(k) == 'a') {
        n= n+1;
    }
}
// postcondition: n is the number of times 'a' occurs in s

E18. This solution looks only for lowercase vowels
int numVowels= 0;
int i= –1;
// invariant: numVowels is the number of vowels in s[0..i]
while (i != s.length()-1) {
    i= i + 1;
    if (s.charAt(i) == 'a' || s.charAt(i) == 'e' || s.charAt(i) == 'i' || s.charAt(i) == 'o' || s.charAt(i) == 'u')
        numVowels= numVowels + 1;
}
// postcondition: numVowels is the number of vowels in s[0..s.length()-1]

E19. int adjEqChars= 0;
int i= 0;
// invariant: adjEqChars is the number of adjacent equal characters in s[0..i]
while (i < s.length()-1) {
    if (s.charAt(i) == s.charAt(i+1))
        adjEqChars= adjEqChars + 1;
    i= i + 1;
}
// postcondition: adjEqChars is the number of adjacent equal characters in s[0..s.length()-1]

E20. int i= 0;
// invariant: the characters in s[0..i–1] are in descending order
while (i < s.length()-1 && s.charAt(i) >= s.charAt(i+1)) {
    i= i + 1;
int i = s.length() - 1;
// invariant: s[i+1..s.length()-1] is all blanks
while (i >= 0 && s.charAt(i) == ' ')
    i = i - 1;
// postcondition: i is the number of blanks at the end of s

int h = 0;
int k = s.length() - 1;
// invariant: s[0..h-1] is the reverse of s[k+1..s.length()-1]
while (h < k && s.charAt(h) == s.charAt(k))
    h = h + 1; k = k - 1;
// postcondition: s[0..h-1] is reverse of s[k+1..s.length()-1] and
// either h >= k or s.charAt(h) == s.charAt(k)
boolean b = (h >= k);  // b = "s is a palindrome"

int k = 0;
boolean b = true;
// inv: b = "every character in s[0..k-1] has the same char next to it in s"
while (b && k < s.length())
    // Set before to "s[k] has the same character before it"
    boolean before = k != 0 && s.charAt(k) == s.charAt(k - 1);
    // set after to "s[k] has the same character after it"
    boolean after = k + 1 < s.length() && s.charAt(k) == s.charAt(k + 1);
    b = before || after;
    k = k + 1;

int k = 1;
boolean b = true;
// invariant: b = "s[c] of s[0..k-1] is a digit for any c that is a power of two
while (b && k < s.length())
    b = b && Character.isDigit(s.charAt(k));
    k = 2 * k;
// postcondition: b = s[c] is a digit for any c that is a power of two

boolean b = s.length() == t.length();
// invariant: b = "s and t have the same length and s[0..k-1] = t[0..k-1]"
for (int k = 0; b && k < s.length(); k = k + 1)
    if (s.charAt(k) != t.charAt(k))
        b = false;
// postcondition: b = (s[] == t)

String t = "";
int k = 0;
// invariant: t is s[0..k-1] but with twins added and
// if s[k-1] has a twin, it is s[k-2]
while (k != s.length())
    // Append two copies of s[k] to t
    t = t + s.charAt(k) + s.charAt(k);
k = k+1;
// if s[k-1] has a twin in s, add 1 to k
if (k < s.length() && s.charAt(k-1) == s.charAt(k)) {
    k = k+1;
}